Compressed Index for a Dynamic Collection of Texts

Ho-Leung Chan, Wing-Kai Hon, and Tak-Wah Lam *

Department of Computer Science
The University of Hong Kong, Hong Kong
{hlchan,wkhon,twlam}@csis.hku.hk

Abstract. Let T be a string with n characters over an alphabet of bounded size. The recent breakthrough on compressed indexing allows us to build an index for T in optimal space (i.e., $O(n)$ bits), while supporting very efficient pattern matching [2, 4]. This paper extends the work on optimal-space indexing to a dynamic collection of texts. Precisely, we give a compressed index using $O(n)$ bits where n is the total length of texts, such that searching for a pattern P takes $O(|P| \log n + occ \log^2 n)$ time where occ is the number of occurrences, and inserting or deleting a text T takes $O(|T| \log n)$ time.

1 Introduction

Indexing a text to support efficient pattern matching has been studied extensively. The recent breakthrough allows us to build an index in optimal space (i.e., $O(n)$ bits), without sacrificing the speed of pattern matching. This paper extends the work of optimal-space indexing to a dynamic setting, i.e., allowing efficient updating and searching a set of texts. The problem we consider is also known as the library management problem [9] in the literature, which is defined as follows: We need to maintain a collection $\mathcal{L}$ of texts; from time to time, a text may be inserted or deleted from $\mathcal{L}$, and a pattern P may be given and its occurrences in $\mathcal{L}$ are to be reported. This problem occurs naturally in the management of homepages [3], DNA/protein sequences [10], and many other real-life applications.

In the static version of the problem, the texts in $\mathcal{L}$ never change. A simple solution is to concatenate all the texts and then build a suffix tree [9, 13] or a suffix array[8]; the searching time for a pattern P with length p is $O(p + occ)$ and $O(p + \log n + occ)$, respectively, where occ is the total number of occurrences of P in $\mathcal{L}$. Note that a suffix tree of a string is a compact trie containing all suffixes of the string, while a suffix array is an array of all suffixes of the string arranged in lexicographical order. Both indexes occupy $O(n \log n)$ bits storage. For indexing the huge amount of web pages or DNA/protein sequences, this space requirement may be too demanding. For example, the suffix tree for the human genome (totally 3G characters) takes 40G bytes of memory, while the suffix array takes 13G bytes [6].

* This work was supported in part by the HKU RGC Grant HKU-7042/02E.

Recently, two exciting results have been made on providing indexes occupying only $O(n)$ bits, yet supporting efficient pattern searching. Essentially, both of them can be considered as another form of the suffix array, storing in a compact manner. The first one is the compressed suffix arrays (CSA) [4] proposed by Grossi and Vitter, which supports pattern searching in $O(p \log n + occ \log^{\epsilon} n)$ time, for any fixed $\epsilon > 0$. The second one is the FM-index [2] by Ferragina and Manzini, with which pattern searching can be done in $O(p + occ \log^{\epsilon} n)$ time. These data structures are also sound in practice. Using CSA or FM-index, one can index the human genome with 1.5G bytes of memory [5].

For a dynamic collection $\mathcal{L}$ of texts, texts can be inserted into or deleted from $\mathcal{L}$. If $O(n \log n)$ bits of space is allowed, one can build a generalized suffix tree (i.e., a single compact trie containing the suffixes of each text in $\mathcal{L}$). Then, to insert or delete a text of length t in $\mathcal{L}$, we update the generalized suffix tree by adding or removing all suffixes of the text, which can be done in $O(t)$ time. For searching a pattern P, the time remains $O(p + occ)$.

To reduce space, one may attempt to 'dynamize' a compressed index such as CSA or FM-index. Indeed, Ferragina and Manzini have demonstrated in [2] how to maintain multiple FM-indexes so as to support a dynamic collection of texts. Their solution requires $O(n+m \log n)$ bits, where m is the number of texts in the collection. Pattern matching is slowed down slightly, using $O(p \log^3 n + occ \log n)$ time. But insertion and deletion has only an amortized performance guarantee; precisely, insertion and deletion of a text of length t take $O(t \log n)$ and $O(t \log^2 n)$ amortized time, respectively. Note that in the worst case, a single insertion or deletion may require re-constructing many of the FM-indexes, using $\Theta(n/\log^2 n)$ time even if t is very small.

Our results. In this paper, we introduce a compressed index for the dynamic library management problem, which requires only $O(n)$ bits. Inserting or deleting a text of length t takes $O(t \log n)$ time, while searching for a pattern takes $O(p \log n + occ \log^2 n)$ time. Note that the time complexities of all operations are measured in the worst case (instead of the amortized case). To our knowledge, this is the first result that requires only $O(n)$ bits, yet supporting both update and searching efficiently, i.e., in $O(t \log^{O(1)} n)$ and $O((p + occ) \log^{O(1)} n)$ time, respectively.

Technically speaking, our compressed index is based on CSA and FM-index. Yet a few more techniques are needed in order to achieve the optimal space requirement and efficient updating. Firstly, recall that the index proposed in [2] requires $O(n + m \log n)$ bits. We have a simple but useful trick in organizing the texts to avoid using a lot of space when the collection involves a lot of very short strings, thus eliminating the $m \log n$ term. Secondly, the original representations of CSA and FM-index do not support updates efficiently. For instance, the index proposed in [2], essentially requires re-building one or more FM-index whenever a text is inserted. Inspired by a dynamic representation of CSA in [7], we manage to dynamize the CSA and the FM-index to support efficient updates to a collection of texts. With either of them, we can immediately obtain an $O(n)$-bit index that supports updates in $O(t \log^2 n)$ time. For pattern matching, using FM-

index alone can achieve $O(p \log n + occ \log^2 n)$ time, and using CSA alone takes $O(p \log^2 n + occ \log^2 n)$ time.

Last but not the least, we find that FM-index and CSA can complement each other nicely to further improve the update time. Roughly speaking, in the process of updating such suffix-array based compressed indexes, we need two pieces of crucial information; we observe that one of them can be provided quickly by FM-index, and the other can be provided quickly by CSA. Thus, by maintaining both CSA and FM-index together, we can perform the update in a straightforward manner, improving the update time to $O(t \log n)$.

Remarks. (1) In the above discussion, we assume that the alphabet Σ has a constant size. For a variable size alphabet, our compressed index occupies $O(n|\Sigma|)$ bits, which may become a problem if $|\Sigma|$ is huge. Nevertheless, our compressed index based on CSA alone achieves a space complexity of $O(n \log |\Sigma|)$ bits. As noted before, update and pattern searching suffer a slowdown by a logarithmic factor. (2) In practice, for indexing a collection of DNA texts, the data structures suggested in this paper occupy $5n$ or $9n$ bits (corresponding to using CSA alone, or using CSA plus FM-index, respectively).

2 Review on CSA and FM-index

Let $T[1..n] = T[1]T[2] \cdots T[n]$ be a string of length n over an alphabet Σ. For any $i = 1, \ldots, n$, $T[i..n]$ is a suffix of T. The suffix array $\texttt{SA}[1..n]$ of T is an array of integers such that $T[\texttt{SA}[i]..n]$ is lexicographically the i-th smallest suffix of T.

The main component of CSA is the function $\Psi[1..n]$ where $\Psi[i] = \texttt{SA}^{-1}[\texttt{SA}[i]+1]$. Let i be the lexicographical order of the suffix $T[\texttt{SA}[i]..n]$. $\Psi[i]$ gives the lexicographical order of the suffix $T[\texttt{SA}[i]+1..n]$. Unlike $\texttt{SA}$, Ψ admits an $O(n)$-bit representation and $O(1)$-time retrieval. We can count the number of occurrences of a pattern P in T using $O(|P| \log n)$ queries to Ψ [4].

The main component of FM-index is the function *count*, which is defined based on the $\texttt{BWT}$ array [1]. For $i = 1, \ldots, n$, $\texttt{BWT}[i]$ is the character $T[\texttt{SA}[i] - 1]$. For each character $c \in \Sigma$ and $i = 1, \ldots, n$, the function $count(c, i)$ is the number of character c appearing in $\texttt{BWT}[1..i]$. Similar to CSA, $count(c, i)$ admits an $O(n)$-bit representation and $O(1)$-time retrieval, and we can count the number of occurrences of a pattern P in T using $O(|P|)$ queries to *count* [2]. See the figure below for an example of the Ψ, $\texttt{BWT}$ and *count* functions.

$T =$ $ a b b a a a b a

i	suffixes in sorted order	$\texttt{SA}[i]$	$\Psi[i]$	$\texttt{BWT}[i]$	$count(\texttt{a}, i)$	$count(\texttt{b}, i)$
1	$ a b b a a a b a	1	6	a	1	0
2	a	9	1	b	1	1
3	a a a b a	5	4	b	1	2
4	a a b a	6	5	a	2	2
5	a b a	7	7	a	3	2
6	a b b a a a b a	2	9	$	3	2
7	b a	8	2	a	4	2
8	b a a a b a	4	3	b	4	3
9	b b a a a b a	3	8	a	5	3

The two lemmas below give the relationship between the Ψ function of CSA and the *count* function of FM-index.

Lemma 1. *Consider a text $T[1..n]$ over the alphabet Σ. For any $c \in \Sigma$ and $i = 1, \ldots, n$, we can compute count(c, i) using $O(\log n)$ queries to Ψ.*

Proof. Observe that $T[SA[j]] = c$ if and only if $\texttt{BWT}[\Psi[j]] = c$. In other words, *count*(c, i) is the number of j such that $T[\texttt{SA}[j]] = c$ and $\Psi[j] \leq i$.

Note that $T[\texttt{SA}[j]] = c$ means that j is in the consecutive region of SA whose suffixes have the first character c. Thus, *count*(c, i) is the number of j in the region with $\Psi[j] \leq i$.

As shown in [12], the Ψ values are increasing in the above region for j. This implies that *count*(c, i) can be found by a binary search on Ψ in the above region, using $O(\log n)$ queries.

Lemma 2. *Consider a text $T[1..n]$ over the alphabet Σ. For any $i = 1, \ldots, n$, we can compute $\Psi[i]$ using $O(\log n)$ queries to the count function.*

Proof. Consider the suffix $T[\texttt{SA}[i]..n]$. Let c be its first character and y be the lexicographical order of which among all suffixes that begins with c. This implies that $\texttt{BWT}[\Psi[i]] = c$, and $\Psi[i]$ is the y-th entry in BWT which is c. This means that $\Psi[i]$ is the first entry in *count*$(c, \cdot)$ array whose value is y. Thus, $\Psi[i]$ can be found based on binary search on *count*$(c, \cdot)$, using $O(\log n)$ queries.

Next, we state a lemma to demonstrate the searching ability provided by the *count* function.

Lemma 3. ([2]) *Consider a text $T[1..n]$ over an alphabet Σ. Let P be any pattern and let c be any character over Σ. Denote the lexicographical order of P among all suffixes of T (i.e., 1 + the number of such suffixes less than P) as i. Then, count$(c, i-1) + 1 +$ offset$_c$ is the lexicographical order of cP among all suffixes of T, where offset$_c$ denotes the number of suffixes of T starting with a character less than c.*

We refer to an execution of the above lemma a *backward search step*. Applying backward search steps repeatedly, we can find the number of occurrences of any pattern P in T using $O(|P|)$ queries to the *count* function. Such a searching method is also known as the *backward search* algorithm in the literature.

3 High Level Organization

This section gives the high level description of a compressed index for a dynamic collection of texts. In particular, we show how to exploit three dynamic data structures, namely, *COUNT*, *MARK*, and *PSI*, to support efficient pattern matching and text updating. The implementation details will be shown in the next section.

Below we first introduce *COUNT*, which is the core data structure that already supports counting the occurrences of a pattern P efficiently, and quick insertion or deletion of texts. Afterwards, we discuss how to exploit *MARK* and *PSI* to support efficient enumeration of the positions where a pattern P occurs, and further speed up the updating process.

Consider a set of texts $\mathcal{L} = \{T_1, T_2, \ldots, T_m\}$ over a finite alphabet Σ. We assume that the texts are distinct, and each text T starts with a special character '\$' in Σ, which is alphabetically smaller than all other characters in Σ and '\$' does not appear in any other part of a text. Denote the total length of texts as n. We always label (relabel) the existing texts in $\mathcal{L}$ in such a way that T_j refers to the lexicographically j-th texts currently in $\mathcal{L}$.

Conceptually, we want to construct a suffix array `SA` for the texts by listing out all suffixes of all texts in lexicographical order. For $i = 1, 2, \ldots, n$,

$$\texttt{SA}[i] = (j, \ell)$$

if the suffix $T_j[\ell..]$ is the lexicographically i-th suffix. To insert a text T to $\mathcal{L}$, we insert all suffixes of T into the `SA`. Similarly, to delete a text from $\mathcal{L}$, we delete all suffixes of T from `SA`. Searching for a pattern P is done by determining the interval $[x, y]$ such that each suffix from $\texttt{SA}[x]$ up to $\texttt{SA}[y]$ has P as a prefix. $\texttt{SA}[x], \texttt{SA}[x+1], \ldots, \texttt{SA}[y]$ gives all locations where P occurs in $\mathcal{L}$. However, the `SA` table needs $O(n \log n)$ bits.

3.1 Basic Data Structure

Due to the space restriction, we cannot directly store the `SA` table. Instead, we use the FM-index, which requires only $O(n)$ bits, to represent the `SA` table implicitly. Recall that FM-index requires the function $count(c, i)$ which returns the total number of occurrences of character c in $\texttt{BWT}[1..i]$. We implement the $count(c, i)$ function with a dynamic data structure *COUNT* and its performance is summarized in the lemma below. The proof (i.e., the detailed construction of *COUNT*) will be given in Section 4.

Lemma 4. *We can maintain the COUNT data structure using $O(n)$ bits space such that each of the following operations is supported in $O(\log n)$ time.*

- *Report(c, i): Returns the value of count(c, i).*
- *Insert(c, i): Updates count$(\cdot,\cdot)$ due to a character c inserted to position i of* `BWT`.
- *Delete(i): Updates count$(\cdot,\cdot)$ due to a character deleted from position i of* `BWT`.

Note that *COUNT* allows efficient updates which is needed when the `BWT` array changes due to insertion or deletion of texts. Our implementation is different from the original one in [2], where the $count(c, i)$ function is stored in a data structure which is difficult to update (but allows constant time query).

Pattern matching. We maintain an array FC such that $FC[c]$ stores the frequency count of the character c in $\mathcal{L}$, which equals the number of suffixes whose first character is c. Together with the $COUNT$ data structure, we can support counting the occurrences of $P[1..p]$ in $\mathcal{L}$ using $O(p \log n)$ time as follows. Firstly, the lexicographical order of $P[p]$ can be computed by $\sum_{c<P[p]} FC[c] + 1$. Then, by Lemma 3, we can find the lexicographical order of $P[p-1..p]$ using one query to the *count* function. The process is repeated, so that eventually, we can find the lexicographical order (say, x) of $P[1..p]$. Similarly, we can find the lexicographical order (say, y) of $P[1..p-1]c$, where c is the character in Σ just greater than $P[p]$. We can easily see that the number of occurrences of P is $y - x$. On the other hand, the whole process requires $O(p)$ queries to the *count* function, and each query takes $O(\log n)$ time. Thus, the total time follows.

Text insertion. To insert a text $T[1..t]$, we insert all the suffixes of T to `SA` implicitly, starting from the shortest one. The lexicographical order of $T[t]$, denoted as i, is 1 + number of suffixes in `SA` that starts with a character less than $T[t]$. Thus, we want to insert $T[t]$ at the i-th row `SA` implicitly. To do so, we insert the character $T[t-1]$ to the i-th position of the `BWT` array, using the function $Insert(T[t-1], i)$ provided by $COUNT$. Then, let i' be the lexicographical order of $T[t-1..t]$, which can be found easily by one backward search step, and we insert $T[t-2]$ to the i'-th position of `BWT`. The process continues until the longest suffix $T[1..t]$ is inserted implicitly to `SA`, which is simulated by inserting $T[1]$ to `BWT`. The whole process takes $O(t \log n)$ time.

Text Deletion. Deleting a text $T[1..t]$ from the collection of texts is more troublesome because among all those suffixes that is a single character $T[t]$, we do not know which one belongs to T. To handle the problem, we perform a backward search for $T[1..t]$ and let $[x, y]$ be the interval such that for any $i \in [x, y]$, $T[1..t]$ is a prefix of `SA`$[i]$. Recall that all texts in the collection are distinct and each of them starts with a special character '\$' which is alphabetically smaller than all other characters. Thus, we can conclude that `SA`$[x]$ corresponds to the text $T[1..t]$ to be deleted because no other text can have $T[1..t]$ as a prefix and have lexicographical order less than $T[1..t]$. Then, we delete all suffixes of $T[1..t]$, starting from the longest one. Note that if the Ψ function of CSA is given, we can determine the lexicographical order of $T[2..t]$ easily from the lexicographical order $T[1..t]$. However, as the Ψ function is not available, we need to simulate each query to Ψ by $O(\log n)$ queries to the *count* function. Since a query to *count* takes $O(\log n)$ time, deleting each suffix of $T[1..t]$ takes $O(\log^2 n)$ time, and the whole process takes $O(t \log^2 n)$ time.

Summarizing the discussion, we have the following theorem.

Theorem 1. *Let $\mathcal{L} = \{T_1, T_2, \cdots, T_k\}$ be a set of k distinct strings over a finite alphabet Σ. Let n be the total length of all strings in $\mathcal{L}$. We can maintain $\mathcal{L}$ in $O(n)$-bit space such that counting the occurrences of a pattern $P[1..p]$ takes $O(p \log n)$ time, inserting a text $T[1..t]$ takes $O(t \log n)$ time, and deleting a text $T[1..t]$ takes $O(t \log^2 n)$ time.*

3.2 Additional Data Structures

The basic data structure in the previous discussion does not support retrieving $\texttt{SA}[x]$ efficiently and thus cannot report the positions where a pattern occurs. In the following, we give an additional data structure for such a purpose. Afterwards, we introduce another data structure that is based on Ψ to speed up the deletion process.

Enumerating Pattern Occurrences To compute $\texttt{SA}[x]$ for any given x efficiently, we use the data structure *MARK* to selectively store some of the entries of SA as follows: *MARK* stores the entries $\texttt{SA}[i] = (j, \ell)$ whenever $\ell > 0$ is an integral multiple of $\log n$.

Recall that we require all texts in $\mathcal{L}$ to start with the '\$' character which is lexicographically smaller than any other character in Σ. As a result, for a set of m texts, the first m entries of SA corresponds to the m texts sorted in lexicographical order. Now, suppose that given a certain x, we want to find the value of $\texttt{SA}[x]$ (which is actually (j, ℓ)). We first check whether $\texttt{SA}[x]$ is stored in *MARK*. If so, we obtain the value of $\texttt{SA}[x]$ immediately. Otherwise, we check whether $x \leq m$, which would imply that $\texttt{SA}[x] = (x, 1)$. If both cases are false, we can determine the lexicographical order of the suffix $T_j[\ell-1..]$, denoted as x', easily using backward search with the *COUNT* data structure. We check whether the entry $\texttt{SA}[x']$ is stored in *MARK* or $x' \leq m$. The process continues and after $k \leq \log n$ steps, we will either meet a suffix $T_j[\ell-k..]$ such that $\ell-k$ is a multiple of $\log n$, or $\ell - k = 1$. In both cases, the value of (j, ℓ) can be found accordingly. As shown in Lemma 5, for any value x, testing whether *MARK* stores the tuple $\texttt{SA}[x]$ takes $O(\log n)$ time. Thus, it takes $O(\log^2 n)$ time to find the value of $\texttt{SA}[x]$ for any value x.

Note that *MARK* should allow updates easily when texts are inserted to or deleted from the dynamic collection of texts. In particular, when a suffix is inserted to or deleted from SA, originally stored tuples (j, ℓ) corresponds to different SA entries. For example, when a suffix is inserted at position u of SA, a tuple (j, ℓ), corresponding to $\texttt{SA}[v]$ originally, becomes the a tuple corresponding to $\texttt{SA}[v+1]$ if $v > u$. That is, we need to update the correspondence for tuples and SA values whenever a suffix is inserted or deleted from SA. We summarize the performance of *MARK* in the lemma below. Note that *MARK* stores at most $\frac{n}{\log n}$ entries of SA. We give the actual construction of *MARK* in Section 4.

Lemma 5. *We can maintain a data structure MARK in $O(n)$ bits such that it stores a tuple $(i, (j, \ell) = \texttt{SA}[i])$ for all entries $\texttt{SA}[i]$ whenever ℓ is a multiple of $\log n$. MARK supports each of the following operations in $O(\log n)$ time.*

- *Report(i): Returns the $(i, (j, \ell))$ if this tuple is stored. Else, return false.*
- *Insert(i, j, ℓ): Inserts the tuple $(i, (j, \ell))$ to MARK.*
- *Delete(x): Deletes the tuple $(i, (j, \ell))$ from MARK.*
- *Increment_lexico(k): For each tuple stored, the j value is incremented by one if the original j value is at least k. This function allows us to update the*

lexicographical order of the texts after a new text with lexicographical order k is inserted.
- *Decrement_lexico(k): For each tuple stored, the j value is decrement by one if the original j value is greater than k.*
- *Shift_up(k): For each tuple stored, the i value is incremented by one if the original i value is at least k. This function allows us to update the correspondence between tuples and* `SA` *after a suffix is insert to position k of* `SA`.
- *Shift_down(k): For each tuple stored, the i value is decremented by one if the original i value is greater than k.*

With *COUNT* and *MARK*, we can find the positions where a pattern $P[1..p]$ occurs in the collection of texts in $O(p \log n + occ \log^2 n)$ time.

Speeding Up the Deletion Recall that to delete a text $T[1..t]$, we first determine the location of $T[1..t]$ in `SA`. Then, we delete all the suffixes of T starting from the longest one. The bottleneck for the deletion operation is determining the lexicographical order of $T[r+1..t]$ after the deletion of the suffix $T[r..t]$. We observe that CSA provides a good solution for it. In fact, the Ψ function of CSA stores exactly the information we needed.

However, we cannot use the original implementation of Ψ as we need to update Ψ efficiently. We dynamize Ψ with the data structure *PSI*, whose performance is summarized in the lemma below. Recall that Ψ is a list of n integers defined as follows. Given an integer i, let $S[r..s]$ is the lexicographically i-th suffix in `SA`. $\Psi[i]$ is the lexicographical order of the suffix $S[r+1..s]$ in `SA`.

Lemma 6. *We can maintain the PSI data structure in $O(n)$ bits such that each of the following operations can be done in $O(\log n)$ time.*

- *Report(i): Return $\Psi(i)$.*
- *Insert(i, x): Insert the integer x to position i of the list of integers. This function is needed when we insert a suffix to* `SA`.
- *Delete(i): Delete the integer from position i of the list.*
- *Shift_up(k): Increment by one for each integer in the list with value at least k. This function is needed when we insert a suffix to position i of* `SA`.
- *Shift_down(k): Decrement by one fore each integer in the list with value greater than k.*

With the *PSI* data structure, insertion and deletion of a text of length t can both be done in $O(t \log n)$ time.

3.3 Summary

We summarize how the search, insert and delete operations are performed with *COUNT*, *MARK*, and *PSI*.

Searching for a pattern $P[1..p]$. We perform backward search to determine the interval $[x, y]$ such that for each $i \in [x, y]$, `SA`$[i]$ corresponds to an occurrence

of P. This can be done in $O(p \log n)$ time using the *COUNT* data structure. Then, for each $i \in [x, y]$, the value of $\texttt{SA}[i]$ is obtained by at most $\log n$ backward search steps, with one query to *MARK* in each step. Thus, the time is $O(p \log n + occ \log^2 n)$.

Inserting a text $T[1..t]$. Intuitively, we insert each suffix of T to SA starting from the shortest one. For $r = t, t-1, \ldots, 1$, we first determine the lexicographical order of $T[r..t]$ in SA and let that value be k. To simulate the effect of inserting $T[r..t]$ into position k of SA, we need to update *COUNT* by inserting $T[r-1]$ to position k of BWT. Then, we need to update *PSI* by incrementing all integers in *PSI* whose value at least k, followed by inserting the lexicographical order of $T[r+1..t]$ to the i-th position of *PSI*. We also need to increment the i value for any tuples in *MARK* whose i value is at least k. Note that the longest suffix will be inserted to some position x of SA, where x is the lexicographical order of T in the collection of texts. Therefore, for each tuple in *MARK* with j value at least x, we increment its j value by one. Finally, we insert tuples corresponding to T to *MARK*. The total time required is $O(t \log n)$.

Deleting a text $T[1..t]$. Intuitively, we delete each suffix of T starting from the longest one. We first determine the lexicographical order of T in the collection of texts. Afterwards, the lexicographical order of the other suffixes of T can be found using the *PSI*. Updating of *COUNT*, *MARK*, and *PSI* are done similarly to that of inserting a text, except that we are decrementing the values this time. The total time required is $O(t \log n)$ as well.

Adjustment due to huge updates. Note that in the above discussion, our data structures require the value of $\log n$ as a parameter, and we have assumed that this value is fixed over the time. This is not true in general as texts can be inserted or deleted in the collection. Thus, when the value of $\log n$ changes, our data structures become different. A simple way to handle this is to reconstruct everything when necessary, but this would imply huge update time, say, $O(n)$ time, on the single update operation that induces the change. Nevertheless, using a standard technique for global rebuilding [11], we can distribute the reconstruction process over each update operation, so that we can bound the update time to be $O(t \log n)$, while having a new data structure ready when $\log n$ is changed. Details will be shown in the full paper.

Summarizing the results, we have the following theorem.

Theorem 2. *Let $\mathcal{L} = \{T_1, T_2, \cdots, T_k\}$ be a set of k distinct strings over a finite alphabet Σ. Let n be the total length of all strings in $\mathcal{L}$. We can maintain $\mathcal{L}$ in $O(n)$-bit space such that inserting or deleting a text $T[1..t]$ takes $O(t \log n)$ time and searching for a pattern $P[1..p]$ takes $O(p \log n + occ \log^2 n)$ time, where occ is the total of occurrences.*

4 Implementing the Data Structures

In this section, we explain how each of data structure *COUNT*, *MARK*, and *PSI* are implemented.

4.1 *COUNT*

Recall that the *COUNT* data structure maintains the function $count(c, i)$, which returns the total number of occurrences of the character c in $\texttt{BWT}[1..i]$. To implement *COUNT*, we store $|\Sigma|$ lists of bits, denoted as $COUNT_c$ for each $c \in \Sigma$. Each list is n bits long and $COUNT_c[i] = 1$ if $\texttt{BWT}[i] = c$ and $COUNT_c[i] = 0$ otherwise.

To support updates easily, for each list $COUNT_c$, we partition it into segments of $5 \log n$ to $10 \log n$ bits long. The segments are stored in a red-black tree, so that a left to right traversal of the tree gives the list $COUNT_c$. Precisely, each node u in the tree contains the following fields.

- A color bit (red or black), a pointer to parent, a pointer to the left child and a pointer to the right child.
- A segment of bits, with length $5 \log n$ to $10 \log n$.
- An integer *size* indicating the total number of bits contained in the subtree rooted at u.
- An integer *sum* indicating the total number of 1 contained in the subtree rooted at u.

To support the function $count(c, i)$, we search the tree of $COUNT_c$ for the node u that contains the i-th bit. We record the number of 1's in the segment of u preceding the i-th bit, and also the *sum* of the left child of u. Then, we traverse from u to the root. For every left parent v on the path, we record the number of 1's in the segment of v and also the *sum* of the left child of v. Summing up all these recorded values gives the number of 1's in the list of $COUNT_c$ up to the i-th bit, which equals $count(c, i)$. The whole process takes $O(\log n)$ time.

To update the *COUNT* data structure when a character c is inserted to position i of BWT column, we insert a bit 1 to position i of $COUNT_c$ and insert a bit 0 to position i of $COUNT_{c'}$ for each $c' \neq c$. The time required is $O(\log n)$. Deletion of a character from BWT can be done in the opposite way in $O(\log n)$ time.

For the space requirement, we note that each node takes $O(\log n)$ bits and there are $O(\frac{n}{\log n})$ nodes. Thus, the space requirement is $O(n)$ bits.

4.2 *MARK*

Recall that *MARK* is a set of at most $\frac{n}{\log n}$ tuples, each in the format of (i, j, ℓ). Note that no two tuples have the same i value, but there may be more than one tuple having the same j value.

To support efficient update, we maintain two red-black trees, one for the i values and the other for the j and ℓ values as follows.

For all the i values of the tuples, they are stored in a red-black tree R_i, such that the left to right traversal of the tree gives the i values of the tuples stored in sorted order.

For all the j values of the tuples (allowing duplication), they are stored in a red-black tree, denoted as R_j, such that the left to right traversal of the tree gives the j values of all the tuples stored in sorted order.

Let $i(u)$ be the i value stored in the node u in R_i. Let $j(v)$ be the j value stored in the node v in R_j. To represent the tuples, we store a pointer at each node u in R_i, pointing to a node v in R_j if $i(u)$ and $j(v)$ belongs to the same tuple. Furthermore, the ℓ value of the tuple $(i(u), j(v), \ell)$ is stored in the node v in R_j.

More precisely, each node in R_i has the following fields.

- A color bit (red or black), a pointer to the left child and a pointer to the right child.
- An integer $\mathit{diff}(u) = i(u) - i(lp(u))$, where $lp(u)$ denotes the left parent of u. $i(lp(u)) = 0$ if $lp(u)$ does not exist.
- A pointer to a node v in R_j.

Each node in R_j has the following fields.

- A color bit (red or black), a pointer to the left child and a pointer to the right child.
- An integer $\mathit{diff}(v) = j(v) - j(lp(v))$, where $lp(v)$ denotes the left parent of u. $j(lp(v)) = 0$ if $lp(v)$ does not exist.
- A pointer to a node u in R_i.
- An integer ℓ.

Although we do not store the value $i(u)$ explicitly for every node $u \in R_i$, its value can be recovered when we traverse down the tree R_i starting from the root. More precisely, when we traverse down the tree, for every node x we meet on the path, we can compute the values $lp(x)$ and $i(x)$ in constant time, as follows. Let x' be the parent of x and assume that $lp(x')$ and $i(lp(x'))$ are known. If x is the left child of x', $lp(x) = lp(x')$. Else, $lp(x) = x'$. In both cases, $i(x) = i(lp(x)) + \mathit{diff}(x)$.

Note that R_i and R_j are very similar to a red-black tree and they inherit the advantages of a balanced binary search tree. Searching, inserting and deleting a tuple can be done easily in $O(\log n)$ time.

For any integer k, let $X_k = \{u | u \in R_i \text{ and } i(u) \geq k\}$. Recall that we need to support the function that given k, we increment $i(u)$ by 1 for all $u \in X_k$. We observe that the actual value of $i(u)$ is not stored for every $u \in R_i$. Instead, we store $\mathit{diff}(u) = i(u) - i(lp(u))$. Thus, if the value $i(lp(u))$ is increased by 1, the value $i(u)$ is also increased by 1 automatically. To increment $i(u)$ by 1 for all $u \in X_k$, we search R_i for the node u with smallest $i(u)$ such that $i(u) \geq k$. Then, for any node $w \neq u$ on the path from root to u, we increment the value $\mathit{diff}(w)$ by 1 if w is a right parent of some other node on the path. We increment $\mathit{diff}(u)$ by 1 if u is the left child of some other node on the path. It can be verified that $i(u)$ is incremented by 1 for all $u \in X_k$. The process takes $O(\log n)$ time.

4.3 *PSI*

The *PSI* data structure maintains the Ψ function where for $i = 1, 2, \ldots, n$, $\Psi[i] = \mathtt{SA}^{-1}[\mathtt{SA}[i] + 1]$. As shown in [12], $\Psi[1..n]$ can be partitioned into $|\Sigma|$ increasing sequences. We store each sequence in a separate data structure PSI_c for each $c \in \Sigma$. To support easy insertion or deletion of integers, each PSI_c is based on a red-black tree. The exact construction is similar to that in [7]. We omit the proof here.

References

1. M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Technical Report 124, Digital Equipment Corporation, Paolo Alto, California, 1994.
2. P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Proceedings of Symposium on Foundations of Computer Science*, pages 390–398, 2000.
3. The Google Homepage Search Engine. `http://www.google.com/`.
4. R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Tree with Applications to Text Indexing and String Matching. In *Proceedings of Symposium on Theory of Computing*, pages 397–406, 2000.
5. W. K. Hon, T. W. Lam, W. K. Sung, W. L. Tse, C. K. Wong and S. M. Yiu. Practical Aspects of Compressed Suffix Arrays and FM-index in Searching DNA Sequences. To appear in *Proceedings of Workshop on Algorithm Engineering and Experiments*, 2004.
6. S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software Practice and Experience*, 29(13):1149–1171, 1999.
7. T. W. Lam, K. Sadakane, W. K. Sung, and S. M. Yiu. A Space and Time Efficient Algorithm for Constructing Compressed Suffix Array. In *Proceedings of International Conference on Computing and Combinatorics*, pages 401–410, 2002.
8. U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
9. E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
10. H. W. Mewes and K. Heumann. Genome Analysis: Pattern Search in Biological Macromolecules. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 261–285, 1995.
11. M. H. Overmars. The Design of Dynamic Data Structures. Lecture Notes in Computer Science 156, pages 34–35, 1983.
12. K. Sadakane. Compressed Text Databases with Efficient Query Algorithms based on Compressed Suffix Array. In *Proceedings of International Symposium on Algorithms and Computation*, pages 410–421, 2000.
13. P. Weiner. Linear Pattern Matching Algorithm. In *Proceedings of Symposium on Switching and Automata Theory*, pages 1–11, 1973.